Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Hash Tables

- A **hash table** is a small(ish) array which simulates a huge array.

- Each key belongs in a **fixed location**.

- Multiple keys will map to the same location – need to resolve “collisions”

- We map keys to array slots using a **hash function**.
Why Hash Tables?

- We'll look at arrays with **direct addressing**
  - $O(1)$ insert, delete, lookup
    ...usually!
- **Tradeoffs:**
  - Keys are out of order
  - Collisions possible
  - $O(n)$ worst case
- Hash tables are a generalization of these arrays
Why Hash Tables?

• Hash tables can handle complex keys (multiple fields) with simple comparisons.

• Hash tables are sometimes more memory efficient than trees.
Why Hash Tables?

• Hash tables are ideal when you need **very fast operations** but the key space is **huge**.

• Hash tables are usually used to implement **dictionaries** (a.k.a maps)
  
  key1 → value1
  key2 → value2
  key3 → value3
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Direct-Addressed Arrays

- Normally, in arrays, each key can go anywhere.

- In a **direct-addressed array**, there is a 1-to-1 relationship between keys and slots
  - Each key can go in only one slot
  - Each slot can hold only one key
Direct-Addressed Arrays

- Normally, in arrays, each key can go anywhere.
- In a direct-addressed array, there is a 1-to-1 relationship between keys and slots.
  - Each key value can go in only one slot.
  - Each slot can hold only one key value.

Since keys can only go in one slot, we don't store them!

We store the values that they map to instead.
Direct-Addressed Arrays

- Suppose a set of keys map to related values:
  
  \[
  \begin{align*}
  2 & \rightarrow v_2 \\
  5 & \rightarrow v_5 \\
  6 & \rightarrow v_6 \\
  11 & \rightarrow v_{11}
  \end{align*}
  \]
Direct-Addressed Arrays

- We allocate an array big enough so that the largest key (11) can be used as an index.
# Direct-Addressed Arrays

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V_2</td>
<td></td>
<td></td>
<td>V_5</td>
<td></td>
<td>V_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V_{11}</td>
</tr>
</tbody>
</table>

2 → v_2  
5 → v_5  
6 → v_6  
11 → v_{11}
Direct-Addressed Arrays

- Each key is associated with a single slot
- Store the value there
- Need a default value to represent “not present”
  - Maybe zero
  - Maybe negative one
  - Maybe null
Direct-Addressed Arrays

- Often, the values are pointers to satellite data
  - null means “not present”
Bit Vector (a.k.a BitMap)

- Sometimes the **only** thing we care about is if the key exists or not
- Use 1 bit per key (boolean[])
  - 8 keys in a single byte

0x261

This slide has an endianness error. Can you find it?
Direct-Addressed Arrays

- Direct-addressed arrays work well so long as:
  - No duplicate keys
  - Relatively dense (most slots are used)
Direct-Addressed Arrays

- But often, the key-space is huge
  - All possible 64-bit numbers
  - All possible String s
  - All possible combinations of multiple fields!
- Relatively few data entries
  - Even 1 billion is “few” if the key space is $2^{64}$!
Direct-Addressed Arrays

- We'd like the advantages of arrays
  - $\Theta(1)$ insert, delete, search

- But applicable to sparse key spaces

- Solution: Use a hash function!
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Hash Functions

- A hash table uses a **hash function** to map **real keys** to **array indices**.
  - The index space is small (by comparison)

- Every hash table uses a different hash function.
  - (There are a few very common ones.)
What is a Hash Function?

- A “hash function” maps a key (might be multiple fields!) to a single integer

\[
\begin{align*}
\text{hash}(2) &= 100 \\
\text{hash}(5) &= 0 \\
\text{hash}(6) &= 37 \\
\text{hash}(11) &= 10
\end{align*}
\]
Example Hash

Observe:
Huge input space mapped to tiny output space

- Example hash results:
  \[
  \begin{align*}
  \text{hash(“Alice”, 12345,17)} &= 195 \\
  \text{hash(“Alice”, 12346,17)} &= 23 \\
  \text{hash(“Alice2”, 12347,17)} &= 102 \\
  \text{hash(“Bob”, 00000,94)} &= 195
  \end{align*}
  \]
Example Hash

Observe:
Very similar inputs give very different hash results.

- Example hash results:
  
  \[
  \text{hash(“Alice”, 12345,17) = 195} \\
  \text{hash(“Alice”, 12346,17) = 23} \\
  \text{hash(“Alice2”,12347,17) = 102} \\
  \text{hash(“Bob”, 00000,94) = 195}
  \]
Example Hash

Observe:
Different inputs can map to identical outputs.

This is called a “collision.”

- Example hash results:
  
  hash(“Alice”, 12345,17) = 195  
  hash(“Alice”, 12346,17) = 23  
  hash(“Alice2”, 12347,17) = 102  
  hash(“Bob”, 00000,94) = 195
Properties of a Good Hash

• Every hash function **MUST:**
  - Be deterministic: Always produce the same key given the same input, every time!

• Most hash functions will:
  - Reduces the size of the key space
  - Give uniform pseudo-random distribution of outputs
    - Even if the inputs are very uniform
  - Be hard to reverse
What is a Hash Table?

• A Hash Table is an array
  - Fixed size
  - Keys map to indices using a hash function

• $\Theta(1)$ to insert, delete, search in the common case

• $\Theta(n)$ to if lots of hash collisions
## Hash Table

- **Keys:** 86, 5, -30, 23

The key space might be very large.

Each key maps to a value.
Hash Table

- Keys: 86, 5, -30, 23

The array is small – the **index space** is far smaller than the **key space**.
We choose a hash function. Modulo is simple, but often works.

Notice that the range of the hash function matches the size of the array.

- **Hash Function**: \( h(n) = n \mod 12 \)
- **Keys**: 86, 5, -30, 23
- **Hash Function:** \( h(n) = n \mod 12 \)
- **Keys:** 86, 5, -30, 23
Hash Table Size

• Your hash table size **must** match the range of your hash function!
  - But most hash functions end with \( \text{"mod m"} \), so that's easy – just change \( m \)

• Large Hash Tables
  - Fewer Collisions, better performance

• Small Hash Tables
  - Less memory
Hash Table Size

- Your hash table size can change
  - Make it smaller when it's (mostly) empty
  - Make it larger when it's (mostly) full
  - (In reality, most programs don't do this)

```java
foreach (element E in old array)
    calculate hash(E)
    store E in proper slot in new array
update hash function for new size
```

Critical Step!
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- **Handling Collisions**
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Handling Collisions

- What if two keys map to the same hash value?
  - This is called “collision”

- **Common Answer:**
  Each slot in the hash table is actually the 'head' pointer of a linked list.
Handling Collisions

hash(100) = 3
hash(101) = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4
Handling Collisions

hash(100) = 3
hash(101) = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4
Handling Collisions

hash(100) = 3
hash(101) = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4
Handling Collisions

When we have a collision, we can insert at the head of the list, so this is an $O(1)$ operation.

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>101</td>
</tr>
</tbody>
</table>

- $\text{hash}(100) = 3$
- $\text{hash}(101) = 7$
- $\text{hash}(102) = 3$
- $\text{hash}(103) = 0$
- $\text{hash}(104) = 4$
Handling Collisions

hash(100) = 3
hash(101) = 7
hash(102) = 3
**hash(103) = 0**
hash(104) = 4
Handling Collisions

\[
\begin{align*}
\text{hash}(100) &= 3 \\
\text{hash}(101) &= 7 \\
\text{hash}(102) &= 3 \\
\text{hash}(103) &= 0 \\
\text{hash}(104) &= 4
\end{align*}
\]
Handling Collisions

**Worst Case**
- $\Theta(n)$ cost to delete or search
- $\Theta(1)$ cost to insert (at head)

**Common Case**
- $\Theta(1)$ cost for everything
Is There a Better Way?

• Advantages of the linked list Hash Table
  - Simple
  - $\Theta(1)$ insert operation
  - Use ordinary Linked List code to implement it

• Disadvantages
  - Memory overhead
  - Allocation time
  - Collisions must be rare
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Open Addressing

• Idea:
  - Instead of a linked list, find alternate slots to put the element into
  - Every element stored directly in the array
  - No need for lists

• Simplest version is “linear probing”
  - Find first available slot after the desired location
We will simulate linear probing to see how it resolves collisions.

We start with the first insertion:
key = 100

This key resides in slot 3.

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>7</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
</tr>
</tbody>
</table>
Open Addressing

hash(100) = 3
\textbf{hash(101)} = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4

Next is key=101

This key resides in slot 7.
Open Addressing

hash(100) = 3
hash(101) = 7
**hash(102) = 3**
hash(103) = 0
hash(104) = 4

Next is key=102

This key would normally go into slot 3, but slot 3 is full. This is a **collision**.

We search forward, and find the first free slot, and put it there.
Open Addressing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>100</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>102</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>101</strong></td>
</tr>
</tbody>
</table>

\[
\text{hash}(100) = 3 \\
\text{hash}(101) = 7 \\
\text{hash}(102) = 3 \\
\text{hash}(103) = 0 \\
\text{hash}(104) = 4
\]

**NOTE:**
Since keys may not be in their “normal” positions, searching for a key requires that we inspect many slots – we keep searching until we find an empty slot.

If the array is densely populated, this may ruin our search performance.
Open Addressing

Next is $\text{key}=103$

This key resides in slot 0.

hash(100) = 3
hash(101) = 7
hash(102) = 3
**hash(103) = 0**
hash(104) = 4
Open Addressing

<table>
<thead>
<tr>
<th>Slot</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>102</td>
</tr>
<tr>
<td>5</td>
<td>104</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>101</td>
</tr>
</tbody>
</table>

Next is \( \text{key}=104 \)

This key would normally reside in slot 4. However that slot is full, so we probe to find the next free one.

hash(100) = 3  
hash(101) = 7  
hash(102) = 3  
hash(103) = 0  
**hash(104) = 4**
Open Addressing

What happens if we delete element 102?

Will we be able to find 104 later?

hash(100) = 3
hash(101) = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4
If we simply delete 102, then searches for 104 will find an empty hole at slot [4] – and report “104 not found.”
Open Addressing

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hash(100) = 3
hash(101) = 7
hash(102) = 3
hash(103) = 0
hash(104) = 4

The solution is to leave a “gravestone” in the slot. We can later fill it with another node...but (typically) not empty it entirely.
Open Addressing

- Works well if:
  - Collisions are rare
  - Array is relatively sparse (to speed searching)
  - Delete is rarely performed

- Can have disastrous performance if many keys map to the same slot (or adjacent slots)
Quadratic Probing

• Just like Linear Probing, except:
  - Use a quadratic function to locate the “next” probe point

• Thus, if multiple keys map to the same slot, their 2\textsuperscript{nd} slot to check will be different
Perfect Hashing

• Two-level hash table
  – Any slots with collisions use 2nd-level hash table to resolve collisions

• Guaranteed $\Theta(1)$

• Must know keys ahead of time
• Expensive to find the optimal solution
• Details too complex for this course
Perfect Hashing

slot 0 → 103
slot 1
slot 2
slot 3
slot 4 → 104
slot 5
slot 6
slot 7 → 101

slot 3 → 100

slot 4 → 102
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- **What Makes a Good Hash Function?**
- Other Uses for Hashing
Notes about Hash Functions

- If two data elements are equal, then so are their hashes.
- If the hashes are different, then so are the data elements!
- Sorting by hash value is **pointless**, since the hash function almost always changes the ordering.

**Therefore:** Hashes are used to detect duplicates, never for sorting data.
Uniformly Pseudo-Random

- **“Uniform”** means:
  
  Every possible hash value is used (roughly) as often as every other. No “hot spots” or rarely-used hash values

- **“Pseudo-Random”** means:
  
  While not formally random (it is deterministic), it appears random to casual inspection.
Reversibility

• “Reversible” means:
  It is possible, given reasonable amount of computing power, to find an input key that matches any hash value.

• “Irreversible” means:
  While theoretically possible to reverse, it is so hard that we can assume that it will never be accomplished.
Why Reversibility Matters

● If a hash function is reversible, then an attacker can formulate input data which forces us to match any hash that they desire

● Usually not a huge deal for hash tables
  – We will use reversible hash functions (cheap!)

● A terrible problem for cryptography, storage
  – Use SHA1, SHA256, etc - “cryptographically secure hashes”
Common Hash Functions

• Modulo
  - Good if the input distribution is uniform
  - Very fast if divisor is power of 2 (bit masking!)

• Bit Slicing (shift right, then modulo)
  - Common if the input is an address
    • High-order bits not uniformly distributed
    • Low-order bits not uniformly distributed
    • But middle are probably (roughly) uniform
Common Hash Functions

- \((ak+b) \mod m\) (\(a, b, m\) are constants)
  - It's also a simple PRNG algorithm!
  - Far from perfect, but often pretty good
  - Invent your own constants, or Google for a suggestion

- Many other options
  - If you don't care about perfection, do whatever
  - If you care about perfection, then don't try to do it yourself! Use a standard algorithm.
Universal Hashing

• A “universal hash function” is one that is provably uniform
  – Might not be uniform over the specific data set that is common for your data!
  – Still, probably pretty good for your data...

• One possible class of universal functions:

\[ h(k) = ((ak+b) \mod p) \mod m \]

\(a, b, p, m\) constants

\(p\) is prime and \(p > m\)
Hashing and Adversaries

• Any reversible hash function is vulnerable to adversaries
  - Somebody attacking your website
  - Somebody trying to crack your password
  - Somebody trying to slow down your computer

• A good hash function is good against almost any attack
  - But if the adversary knows your hash function, they can craft the attack to match it
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing
Other Uses for Hashing

- Remember this?
  If two *hashes* are different, then the *keys* were also definitely different

- How can we use this property in other situations?
Finding Duplicates in a Database

- Imagine:
  - Two tables, each with millions of records
  - You want to find records with matching values for some key
  - Neither table is sorted by the key yet
    - $O(n)$ comparison is possible if both are sorted by the key
    - Naive comparison is $O(n^2)$
Finding Duplicates in a Database

- Hash the key for each record
- Organize records into “bins” based on hash value
- Only compare records in matching bins
- Cost:
  - $O(n)$ to read the records
  - $O(b^2)$ to compare the bins (where $b$ is avg # / bin)
  - More bins – smaller value for $b$!
Storing Passwords

- Need to build a password DB
- Can't store passwords in cleartext
- Encryption also not OK (could be decrypted)

Objective:
- Store the password DB in the clear (visible to everybody) /etc/passwd
- But don't give away passwords
Storing Passwords

- Use a non-reversible hash on each password
- Store the **hash value** in the public file, not the password!
- When user tries to login, hash the password they give, see if it matches.
Storing Passwords

• What if two users have the same password?
  • They would have the same hash!
    – Nobody else would know their password...but each of them would know the other's

• So add a random “salt” to each password
  – Hash “salt+password”
  – Store salt with the hashed password
Attacking Passwords

- Brute force attacks
  - Check every password, until you get a hit
- Dictionary attacks
  - Store a password for every possible salt+hash combination

- These attacks became practical 10 or 20 years ago for ordinary passwords
  - The salt+hash is no longer publicly visible
/etc/passwd on lectura

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
... 
russelll:x:18057:23057: Russell Lee Lewis:/home/russelll:/bin/bash

russelll@lectura:~$ ls -al /etc/passwd /etc/shadow
-r--r--r-- 1 root wheel 228570 Sep 18 13:01 /etc/passwd
-r-------- 1 root wheel 240524 Sep 28 13:37 /etc/shadow
Content-Addressable Storage

- We want to compare two files
  - But they are too large to copy over the network
  - Can we detect duplicates?
Content-Addressable Storage

● Instead of using a small hash, use a huge one
  – SHA1: 160 bits
  – Astronomically improbable that we will ever find a collision in the entire history of the universe!
  – Thus, if two files have the same hash, it is as good as certain that they were the same file.

● Can also be used to compare parts of files
Bloom Filter

• Have a huge data set, many keys
• Want to rapidly check “is key X in the set?”
Bloom Filter

- Hash all keys as you insert them
- Have bins (like the DB example), but a large number of them
- Bins are simply **bits**: set the bit to TRUE if you find a key in that bin.

- To lookup: check if the bit is set
  - If not, then the key **certainly does not** exist
  - If so, then **maybe**
Topic 13: Hashing

- Why Hash Tables?
- Direct-Addressed Arrays
- Hash Tables
- Handling Collisions
- Open Addressing
- What Makes a Good Hash Function?
- Other Uses for Hashing

Summary